An Economic Analysis of Software Reuse©

Dr. Randall W. Jensen
Software Technology Support Center

This article presents a simplified economic analysis of the cost of software reuse. The reuse definition used here includes both commercial off-the-shelf (COTS) and existing software from an upgraded platform. The results are independent of software estimating tools or models. The model used in this analysis relates the cost of software development to the reused software level and the costs of developing and maintaining the software components. COTS software is a special case of reuse described in this article.

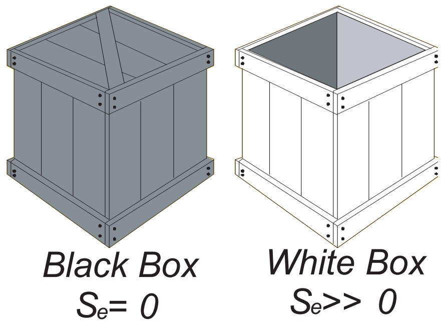
Current software projects tend to maximize reusable component use and minimize development product size. There are significant advantages to using reusable components:

- Lower development time and effort through using existing, supported components.
- Reduced risk through using proven field-tested components.

High customer demand, reduced software development budgets, and a competitive software market drive the need for reusable software. The downside to reusable software is a high development cost, and the significant cost of integrating reusable components into software products. These integration costs can be devastating if components are inadequate, poorly defined and documented, or not quite compatible with the application.

Gaffney and Durek [1] published the first economic analysis report of this type in 1988. Marion Moon and I (while at Hughes Aircraft Company) in 1989 initiated an economic analysis in response to [1]. Unfortunately, the project was shelved before completion due to higher priority tasks. The interest in reuse cost and the need for the economic analysis has continued to increase since that time. Meanwhile, the acronym COTS (commercial off-the-shelf) has largely replaced the term reuse, but the costs associated with reuse have remained the same.

Figure 1: *Black Box Versus White Box*



The analysis in this article focuses on the primary measurable costs associated with reuse, but does not consider several hard-to-predict costs:

- Vendor upgrade release to reusable component.
- Vendor discontinuing component support.
- Component requirement or capability changes.

"High customer demand, reduced software development budgets, and a competitive software market drive the need for reusable software."

- Cost of component evaluation and selection.
- Understanding component function or external interfaces.

The objective of this analysis is to show there are significant cost impacts of software reuse without considering the costs associated with the less-defined factors listed above. The analysis results show significant and somewhat optimistic cost impacts.

Black-Box Phenomenon

The concept of a black box is widely used in system and hardware design. A black box is a system (or component, object, etc.) with known inputs, known outputs, a known input-output relationship, and unknown or irrelevant contents. The box is black; that is, the contents are not visible as shown in Figure 1. The black-box concept is particularly important where components, or objects, are used without the engineering, implementation, integration, and test costs associated with their development.

A white box is a component that requires knowledge of the box contents to be used. A software component becomes a white box when either of the following conditions exist:

- A modification is required to meet software requirements.
- Documentation that is more extensive than an interface or functional description is required before the component can be incorporated into the software system.

Black-box component behavior is characterized in terms of an input set, an output set, and a relationship (hopefully simple) between the two sets. Behavior must be uniquely determined for all input and output combinations. Behavior must be stable and reliable. Once behavior becomes unstable, unreliable, or slightly different than the project needs, the component becomes a white box. Effective size S_e is a major difference between black box and white (or gray) box components from an estimating point of view. The effective size of the software within the black box is zero. Prying the lid off the box has serious consequences in terms of effective size.

Reusable software in this analysis satisfies the black box component definition.

First-Order Reuse Cost Model

There are some conditions we need to assume in this economic analysis:

- The software must satisfy the black-box requirements at the level of abstraction being applied; that is, the reused software satisfies the required performance requirements without modification.
- User knowledge is expert within the scope of reuse.
- Documentation is adequate for the

reuse needs.

- Cost of reusable component selection, evaluation, and purchase are ignored.
- The product is *rock solid*; that is, no maintenance is required, and no vendor upgrades will be made.

A software system contains three categories of source code: new S_n; original S_o, including both modified and *lifted* (lifted is a term for unchanged original code); and reused S_r as shown in Figure 2. The effective size S_e used in most software cost and schedule estimates is an adjusted combination of the new and modified source code similar to the equation:

$$S_e = S_n + S_o (A_d \times F_d + A_i \times F_i + A_t \times F_t) \quad (1)$$

where,

A_d = Design activity, A_i = Integration activity, A_t = Test activity and $A_d + A_i + A_t = 1$. The parenthetical factor is a weighted combination of relative efforts from the design (F_d), implementation (F_i), and test (F_t) activities. More thorough discussions of effective size can be found in the references.

The remainder of the system consists of one or more reusable components. Since reusable components are black boxes that have no accessible size, we cannot directly apply an effective size equation to form an estimate.

For our purposes, we are going to assume the relative reusable component(s) size can be derived by estimating the size of the reusable component built from scratch S_r as:

$$R = \frac{S_r}{(S_e + S_r)} \quad (2)$$

where,

R is the portion (fraction) of the system to be implemented by reusable source code.

The first-level economic model of software reuse begins with the assumption that the cost of software development C for a product relative to the cost of all new source code can be given by the equation:

$$C = 1(1 - R) + bR$$

or

$$C = 1 + R(b - 1) \quad (3)$$

where,

Equation Legend

a	Reusable component development cost relative to the cost of non-reusable development from scratch.
b	Relative cost of incorporating reusable components into developed system.
C	Relative cost of software development.
F	Relative COTS acquisition cost relative to the cost of non-reusable development from scratch.
n	Number of uses over which the reusable product cost is amortized.
R	Portion (fraction) of the system to be implemented by reusable source code.
F_d	Design effort relative to design from scratch.
F_i	Implementation effort relative to implement from scratch.
F_t	Test effort relative to test from scratch.
S_{COTS}	Estimated size of an internally developed COTS replacement.
S_e	Effective source size for devlopment.
S_n	New source code to be added to system.
S_o	Original source size from pre-existing system.

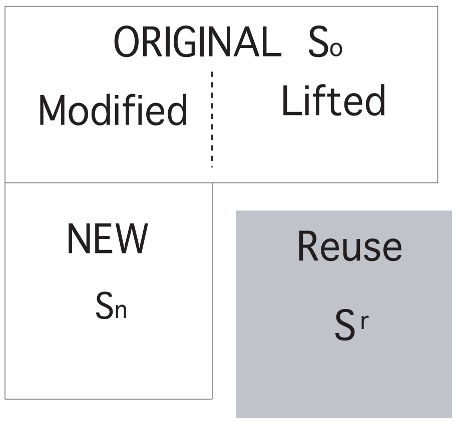
C=1 is the cost of developing a system from scratch. The factor *b* represents the cost of incorporating reused components into the system relative to developing the components from scratch. The term (*1-R*) represents the fraction of new and/or modified source code. This model was first published by Gaffney and Durek [2] and is the basis of this analysis.

Reuse can occur at several levels: requirements, design, code, and validated code. Using the relative design, implementation, and integration factors from Sage [3], we find the relative cost for each development activity given by Table 1 (see page 6). The relative cost values based on Constructive Cost Model (COCOMO) [4]/Revised Intermediate COCOMO (REVIC) [5] are also included in the table for comparison.

The reusable component type (abstraction) determines the relative cost factor *b* in Equation (3). The specific reusable component types considered in this analysis are requirements, design, code, and validated code. The activities to develop these are defined as follows:

- **Requirements.** Includes the analysis and synthesis of software requirements. The product resulting from this activity is the Software Requirements Specification (SRS). The activity is often terminated with a software requirements review (SRR). The definition of system requirements, if present, is not part of the software requirements analysis activity.
- **Design.** Includes the architecture and detailed design of the software product. The resulting product of this activity is a detailed product specification containing both the architecture and component specifications. The activity is usually terminated by a

Figure 2: *Software System Architecture for Reuse Analysis*



Activity	Activity Code	Relative Cost, Sage	Relative Cost, REVIC/COCOMO
Requirements	Req	0.07	0.07
Design	Des	0.38	0.41
Implementation	Imp	0.23	0.26
Integration and Test	Test	0.32	0.26

Table 1: *Relative Costs of Development Activities*

detailed design review (often referred to as a critical design review, or CDR).

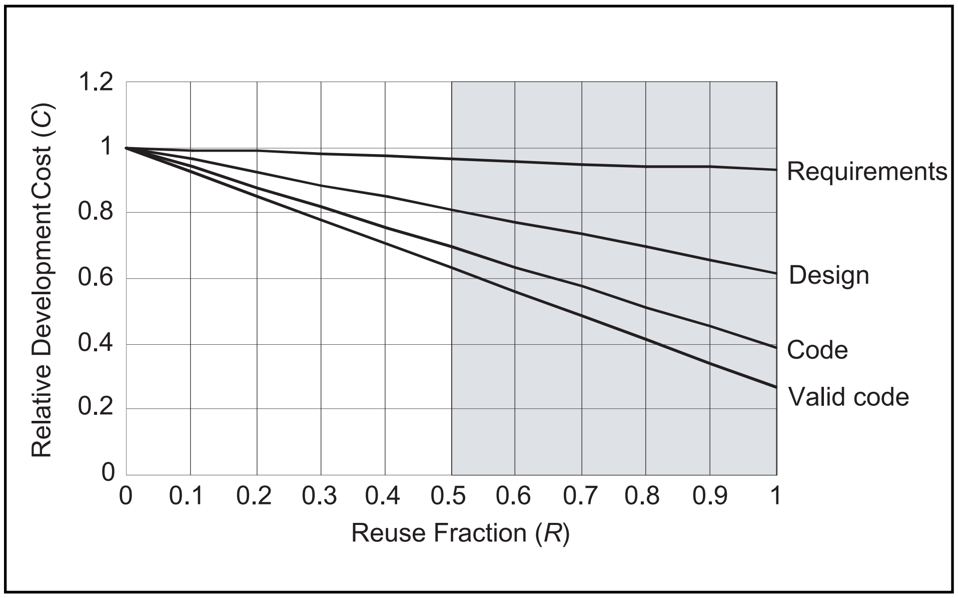
- **Implementation.** Implements the detailed software design in the specified programming language(s), and verifies the individual component (unit) performance to the requirements specified in the detailed product specification.
- **Integration and Test.** Integrates (combines) the tested software components into a larger structure that represents the software product. The activity may contain one or more computer programs. The components are individually defined by formal requirements and interface specifications. The activity usually culminates with a qualification test that evaluates performance per the software requirements specification for the product. The test is usually conducted at the development facility with controlled test data.
- **Regression Test.** Integrates previously validated components into a larger software product structure. This activity may contain one or more computer programs. The regression test activity ends with satisfactory completion of the final qualification test that evaluates product performance per the software requirements and interface specification. Regression test usually reduces the early integration tests required by the integration and test activity.

Table 2 combines the relative activity costs from Table 1 to provide the relative component reuse cost *b* values. For example, the design, implementation, and test activities must be completed to incorporate a requirements reuse component. The incorporation cost is the sum of the relative activity costs, or *b = 0.93*. If the code is reused, the requirements effort for this component of the new system must still be performed. Also note the relative integration cost *b* for validated code is assumed to be *0.27* instead of the *0.32* value assumed by Sage for normal integration and test. This decrease accounts for the reduced testing requirements of validated code.

Table 2: *Relative Reuse Cost*

Component Type	Activities to Be Completed	Relative Reuse Cost (*b*)	Relative Development Cost
Requirements	Design, Implementation, Test	0.93	0.07
Design	Requirements, Implementation Test	0.62	0.45
Code	Requirements, Test	0.39	0.68
Validated Code	Requirements, Test (Regression)	0.27	1.00

Figure 3: *Relative Development Cost vs. Reuse Fraction By Reuse Component Type*



The cost relationship between relative development cost *C* and the percent of reusable software *R* is illustrated graphically in Figure 3. The graph shows the relative costs for each reuse component type (*b* values defined in Table 2). Reuse percentages greater than 50 percent are uncommon and are highlighted with a gray background in the figure. The simple cost model shows that the maximum cost reduction for a software system containing 50 percent COTS software (validated code *b=0.27*) is only about 37 percent (relative development cost is 63 percent). If 100 percent validated code reuse were possible, the software still costs 27 percent of the cost required to build the software system from scratch due largely to regression testing.

Higher Order Cost Model

The first issue that must be considered in developing the reuse cost model is the cost of the reusable component. Incorporating the development cost into the economic model yields:

$$C = (1- R) \times 1 + (b + \frac{a}{n})R \qquad (4)$$

where,

a is the reusable component development cost relative to the cost of non-reusable development from scratch, and *n* is the number of uses over which the reusable product cost is amortized. The model then becomes:

$$C = (b + \frac{a}{n} - 1)R + 1 \qquad (5)$$

The relative component development cost is at least equal to the non-reusable software development cost. The development cost could double when the effort required to make the component more robust is considered. For this analysis we assume the relative component development cost *a* is in the realistic range *$1.0 \leq a \leq 2.0$*. The factor *a/n* in the cost model accounts for the amortized cost of providing the reusable software to this project. Equation (5) shows that as long as the coefficient is $b+a/n \leq 1$, reuse will provide a positive cost incentive; that is, $C \leq 1$. We will look at the cost incentive further in the next section.

The factor *a* can also be used to relate the relative cost of purchasing, or otherwise acquiring, the reusable component(s) for the project. In this case, the component acquisition cost *a* is in the range $0.0 \le a \le 2$ where the reusable component acquisition relative cost includes evaluation, selection, and procurement. As acquisition cost approaches development cost, acquisition becomes less attractive.

The reusable component acquisition cost can be treated in a more conservative manner. Assume the development project is only willing to absorb the amortized cost of the component used in the project. That is, if the project is using only the requirements from the acquired component, we can argue that requirements cost is the only cost to be amortized. In that case, the model becomes:

$$C = (b + \frac{a(1-b)}{n} - 1)R + 1 \qquad (6)$$

where,

$a(1-b)$ represents the requirements acquisition cost. We cannot ignore the cost of maintaining the library of reused components. Let the cost of library maintenance be allocated as a fraction of the component development cost. Incorporating maintenance into Equation (5) we find:

$$C = [b + \frac{a(1-b+d)}{n} - 1]R = 1 \qquad (7)$$

where,

d is the cost fraction added to the component acquisition cost to account for reuse library maintenance. The maintenance fraction value is a function of the size and use of the maintenance library. The maintenance value is also amortized over the number of component uses.

Acquisition Amortization

The reusable component amortization is a function of the number of applications of each component. A large number of reuses *n* reduces the magnitude of the amortization factor *a/n* in each of Equations (5) - (7). The reuse cost coefficient

$$b + \frac{a}{n} - 1 \qquad (8)$$

must be negative in order to provide a cost improvement in Equation (5). Or, in other words, if the coefficient is $b+a/n<1$, the relative software development cost *C* for the project is less than 1.0.

	Relative Cost of Developing Reuse Component (*a*)				
Relative Reuse Cost (*b*)	**1.00**	**1.25**	**1.50**	**1.75**	**2.00**
Requirements (0.93)	15	18	22	25	29
Design (0.62)	3	4	4	5	6
Code (0.39)	2	3	3	3	4
Validated Code (0.27)	2	2	3	3	3

Table 3: *Minimum Reuse Number (n_0) Versus Component Type (b) and Acquisition Cost (a)*

The minimum number of reuse applications can be derived from Equation (8) by setting the coefficient to unity and solving for *n*. The resulting equation shown in Equation (9) represents the number of uses required to cover the reusable component cost. The threshold reuse number (n_0) is:

$$n_0 = \text{ceiling } (\frac{a}{1-b}) \qquad (9)$$

rounded up to the nearest unit. *Ceiling(arg)* is defined as the smallest integer greater than, or equal to, *arg*. The information in Table 3 demonstrates the threshold, or minimum number of reuse applications.

"The models developed in this effort and the results achieved here are independent of software estimating tools or models. This information can be tailored or related to any software cost estimation model."

The threshold reuse number values shown in Table 3 represent the break-even values for reusable component development. The number of reuse applications must be greater than, or equal to, the numbers shown to have a positive impact on the projects using the components.

COTS Cost Model

COTS software is a special application of software reuse. There are several assumptions we must make before specifying the COTS software cost model. The best way to visualize COTS software is as a shrink-wrapped product. This basically means that the software includes the following:

- Contains only validated source code.
- Is purchased and not internally developed or modified.
- Has no library costs associated with the product.
- Conforms to the black-box definition.
- Requires no product maintenance.
- Requires no version upgrades.

The reuse fraction *R* is approximated by estimating the source code size for an internally developed product that is functionally equivalent to the COTS software. The ratio *R* is defined as:

$$R = \frac{S_{COTS}}{S_e + S_{COTS}} \qquad (10)$$

where,

S_{COTS} is the estimated size of an internally developed COTS replacement, and S_e is the effective size of the new, modified, and lifted source code S_o as shown in Figure 2.

Externally developed components (COTS) are simpler to analyze because the development costs are outside the project development environment. Amortization and maintenance costs are still relevant to the economic analysis. The economic cost model for COTS software (validated code) becomes:

$$C = (\frac{F(1+d)}{n} - 0.73)R + 1 \qquad (11)$$

where,

F is the COTS acquisition cost relative to the cost of non-reusable development from scratch. The lower cost limit for free COTS components is approximately 27 percent due to regression testing and software validation. No consideration has been given in Equation (11) to the costs associated with component evaluation and selection, nor has any consideration been allowed for developing expertise in the components' external interface or function.

We can graphically illustrate the relative software costs associated with using COTS software. Let us consider the following example:

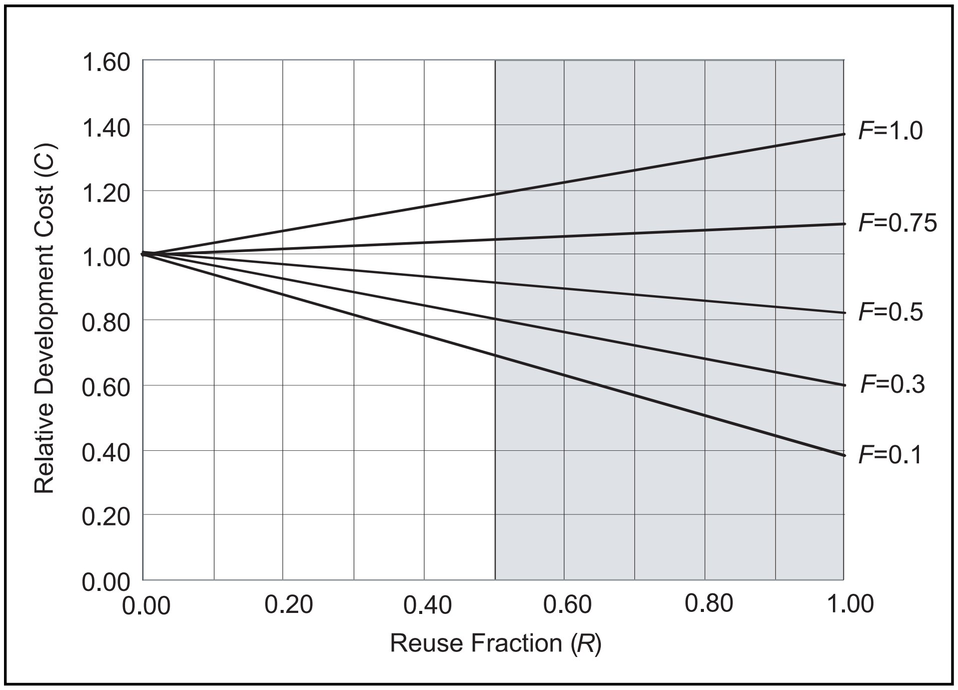


Figure 4: *Relative Development Cost Versus Reuse Fraction By Relative COTS Purchase Cost*

- F is in the practical range $0.1 \leq F \leq 1$; that is, F is limited to the cost of developing the COTS component(s) from scratch.
- Component cost is to be amortized over one (1) application, or $n=1$.
- Maintenance over the useful life of the component(s) is 10 percent, or $d=0.1$.

The relative product software cost C relative to the cost of all new source code calculated from Equation (11) for this example is plotted in Figure 4. The maximum relative acquisition cost F in this model under these conditions to break even is approximately 65 percent of the cost to develop the COTS product from scratch.

If we assume the reusable component is free ($F=0$) and a practical maximum reuse fraction ($R=0.5$), the economic model in Equation (11) shows the relative development cost is approximately 64 percent. The ideal relative development cost with a reuse fraction for $R=0.5$ is 50 percent of the cost of developing the product without reusable components. The economic model prediction is realistically higher than the ideal condition.

Summary and Conclusions

The intent of this effort produced a simplified economic model that provides a realistic prediction of software product development costs in an environment containing reused software components. The reuse definition used in this analysis includes both COTS software and internal software components developed for reuse. The models are developed at two levels. The first level, a truly first-order model, relates relative software product development costs to the fraction of the product to be implemented by reusable components and the reusable component sophistication (requirements, design, etc.). The second level incorporates the significant costs associated with the development, or acquisition of reusable components.

The models developed in this effort and the results achieved here are independent of software estimating tools or models. This information can be tailored or related to any software cost estimation model.

The economic model does not attempt to account for all costs associated with software reuse. The reusable component function and interface complexity issues are ignored here, but are vital estimate elements in practice. There are several cost factors not included because of the difficulty in establishing numeric values for these factors in a broad general sense. These factors, listed in the introduction, should not be ignored in the real application of these models. The factors are major considerations in most projects.◆

References

1. Gaffney Jr., John E., and Thomas A. Durek. "Software Reuse – Key to Enhanced Productivity; Some Quantitative Models." Vers. 1.0. SPC-TR-88-015. Herndon, VA: Software Productivity Consortium, Apr. 1988.
2. Gaffney Jr., 2-1.
3. Software Engineering, Inc. Sage User's Guide. Brigham City, UT: Software Engineering, Inc., May 2001.
4. Boehm, B.W. Software Engineering Economics. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.
5. REVIC Users Group. REVIC Software Cost Estimating Model User's Manual. Vers. 9.0. Arlington, VA: Air Force Cost Center, 1991.

About the Author

Randall W. Jensen, Ph.D., is a consultant for the Software Technology Support Center, Hill Air Force Base, Utah, with more than 40 years of practical experience as a computer professional in hardware and software development. For the past 30 years, he has actively engaged in software engineering methods, tools, quality software management methods, software schedule and cost estimation, and management metrics. He retired as chief scientist of the Software Engineering Division of Hughes Aircraft Company's Ground Systems Group, and was responsible for research in software engineering methods and management. Jensen founded Software Engineering, Inc., a software management-consulting firm in 1980. He developed the model that underlies the Sage and the Galorath Associates, Inc.'s Software Evaluation and Estimation of Resources – Software Estimating Model [SEER-SEM] software cost and schedule estimating systems. Jensen received the International Society of Parametric Analysts Freiman Award for Outstanding Contributions to Paramet-ric Estimating in 1984. He has published several computer-related texts, including "Software Engineering," and numerous software and hardware analysis papers. He has a Bachelor of Science in electrical engineering, a Master of Science in electrical engineering, and a doctorate in electrical engineering from Utah State University.

Software Technology Support Center
6022 Fir AVE, BLDG 1238
Hill AFB, UT 84056-5820
Phone: (801) 775-5742
Fax: (801) 777-8069
E-mail: randall.jensen@hill.af.mil